JAVA 8 Futures:

****

Java 8 came to market in 2014 march 18th.

Java 8 JVM Changes:

==========

Before Java 8 Heap area divided into 3 spaces,

1. Younger Generation
2. Older Generation
3. Permgen Memory

In Permgen are we used to store the static content will be stored and it will also stores the application metadata required by the JVM. The biggest disadvantage of PermGen is that it contains a limited size which leads to an OutOfMemoryError. The default size of PermGen memory is 64 MB on 32-bit JVM and 82 MB on the 64-bit version.

Due to the above problems, PermGen has been completely removed in Java 8. In the place of PermGen, a new feature called Meta Space has been introduced. MetaSpace grows automatically by default.

Metaspace capacity:

==========

By default class metadata allocation is limited by the amount of available native memory (capacity will of course depend if you use a 32-bit JVM vs. 64-bit along with OS virtual memory availability).

A new flag is available (MaxMetaspaceSize), allowing you to limit the amount of native memory used for class metadata. If you don't specify this flag, the Metaspace will dynamically re-size depending of the application demand at runtime.

Metaspace garbage collection

==========

Garbage collection of the dead classes and classloaders is triggered once the class metadata usage reaches the “MaxMetaspaceSize”.

Proper monitoring & tuning of the Metaspace will obviously be required in order to limit the frequency or delay of such garbage collections. Excessive Metaspace garbage collections may be a symptom of classes, classloaders memory leak or inadequate sizing for your application.

Lambda Expressions:

==========

To enable functional programming in java Lambda expressions introduced in Java 8. Lamda Expression is an ananymous function.

Lambda Expressions provide the way to implement the Functional interface.

Lambda Expression Syntax:

(parameters) -> expression

Ex: x-> x+y

```
List<String> pointList = new ArrayList();

pointList.add("1");
pointList.add("2");

pointList.forEach(p ->  { System.out.println(p); } );
```

Rules for writing lambda expressions:

----

i)A lambda expression can have zero, one or more parameters.

ii)The body of the lambda expressions can contain zero, one or more statements.

iii)If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression. When there is more than one statement in body than these must be enclosed in curly brackets.

Advantages of Lamda Expressions:

****

1. Functional Programming.

2. Java lambda expression is treated as a function, so compiler does not create .class file.

3. It provides a clear and concise way to implement SAM interface(Single Abstract Method) by using an expression. It is very useful in collection library in which it helps to iterate, filter and extract data.

==========

Functional Interface:

==========

An interface with only one abstract method isknown as Functional Interface. It can have any number of default and static methods. It can also declare methods of object class.

In Java 8, functional interfaces can be represented using lambda expressions, method reference and constructor references as well.

Ex:

```
@FunctionalInterface
public interface Runnable {
   public abstract void run();
```

}

Do's and Don't's in functional interfaces:

==========

1. In FunctionInterface only one abstract method is allowed in any functional interface. Second abstract method is not not permitted in a functional interface. If we remove @FunctionInterface annotation then we are allowed to add another abstract method, but it will make the interface non-functional interface.

2. A functional interface is valid even if the @FunctionalInterface annotation would be omitted. It is only for informing the compiler to enforce single abstract method inside interface.

3. Conceptually, a functional interface has exactly one abstract method. Since default methods have an implementation, they are not abstract. Since default methods are not abstract you're free to add default methods to your functional interface as many as you like.

4. Another important point to remember is that if an interface declares an abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface's abstract method count since any implementation of the interface will have an implementation from java.lang.Object

==========
==========

Predefined Functional Interfaces:

==========

Java provides predefined functional interfaces to deal with functional programming by using lambda and method references.

Predicate:

****

It is a functional interface which represents a predicate (boolean-valued function) of one argument. It is defined in the java.util.function package and contains test() a functional method.

```
public class PredicateInterfaceExample {
  public static void main(String[] args) {
    Predicate<Integer> pr = a -> (a > 18); // Creating predicate
    System.out.println(pr.test(10));    // Calling Predicate method
  }
```

Methods:

----

test: It evaluates this predicate on the given argument

and: It will be use in predicate chain if first predicate is false, then the other predicate is not evaluated.

negate: It returns a predicate that represents the logical negation of this predicate.

or: It will be use in predicate chain if first predicate is true, then the other predicate is not evaluated.

isEqual: It returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object).

Bi-Predicate:

****

BiPredicate<T,U> It represents a predicate (boolean-valued function) of two arguments.

Function:

****

It is a functional interface. It is used to refer method by specifying type of parameter. It returns a result back to the referred function.

```
public class FunctionInterfaceExample {
```

```
    static Integer addList(List<Integer> list){
        return list.stream()
                .mapToInt(Integer::intValue)
                .sum();
    }
    public static void main(String[] args) {
        // Creating a list and adding values
        List<Integer> list = new ArrayList<Integer>();
        list.add(10);
        list.add(20);
        list.add(30);
        list.add(40);
        // Referring addList() method
        Function<List<Integer>, Integer> fun = FunctionInterfaceExample::addList;
        // Calling Function interface method
        int result = fun.apply(list);
        System.out.println("Sum of list values: "+result);
    }
```

Methods:

----

andThen: It returns a composed function that first applies this function to its input, and then applies the after function to the result. If evaluation of either function throws an exception, it is relayed to the caller of the composed function.

identity: It returns a function that always returns its input argument.

apply: It applies this function to the given argument.

compose: It Returns a composed function that first applies the before function to its input, and then applies this function to the result. If evaluation of either function throws an exception, it is relayed to the caller of the composed function.

Bi-Function:

****

BiFunction<T,U,R> It represents a function that accepts two arguments and returns a a result.

Consumer:

****

It is a functional interface defined in java.util.function package. It contains an abstract accept() and a default andThen() method. It can be used as the assignment target for a lambda expression or method reference.

The Consumer Interface accepts a single argument and does not return any result.

```
public class ConsumerInterfaceExample {
   static void addList(List<Integer> list){
      // Return sum of list values
      int result = list.stream()
            .mapToInt(Integer::intValue)
            .sum();
      System.out.println("Sum of list values: "+result);
   }
   public static void main(String[] args) {
      // Creating a list and adding values
         List<Integer> list = new ArrayList<Integer>();
         list.add(10);
         list.add(20);
         list.add(30);
```

```
        list.add(40);
    // Referring method to String type Consumer interface
    Consumer<List<Integer>> consumer = ConsumerInterfaceExample::addList;
    consumer.accept(list);  // Calling Consumer method

  }
```

Methods:

----

accept: It performs this operation on the given argument.

andThen: It returns a composed Consumer that performs, in sequence, this operation followed by the after operation. If performing either operation throws an exception, it is relayed to the caller of the composed operation. If performing this operation throws an exception, the after operation will not be performed.

Bi-Consumer:

****

BiConsumer Interface accepts two input arguments and does not return any result. This is the two-arity specialization of Consumer interface. It provides a functional method accept(Object, Object) to perform custom operations.

```
public class BiConsumerInterfaceExample {
  static void ShowDetails(String name, Integer age){
    System.out.println(name+" "+age);
  }
  public static void main(String[] args) {
    // Referring method
    BiConsumer<String, Integer> biCon = BiConsumerInterfaceExample::ShowDetails;
    biCon.accept("Rama", 20);
```

```
        biCon.accept("Shyam", 25);
        // Using lambda expression
        BiConsumer<String, Integer> biCon2 = (name, age)->System.out.println(name+" "+age);
        biCon2.accept("Peter", 28);
    }
}
```

Supplier:

****

If a consumer “consumes” your input, a supplier “supplies” you an output. The Supplier interface defined in the java.util.function package has a single abstract method get that doesn’t accept any argument and returns an object of type <T>

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

Methods:

----

get: this methods will return the result.

==========

Method Reference:

==========

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference. In this tutorial, we are explaining method reference concept in detail.

In Java 8, we can refer a method from class or object using class::methodName type syntax.

Types of Method References

==========

There are following types of method references in java:

Reference to a static method.

Reference to an instance method.

Reference to a constructor.

Reference to a static method.:

==========

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

Syntax : ContainingClass::staticMethodName

```
interface Sayable{
   void say();
}
public class MethodReference {
   public static void saySomething(){
      System.out.println("Hello, this is static method.");
   }
   public static void main(String[] args) {
      // Referring static method
      Sayable sayable = MethodReference::saySomething;
      // Calling interface method
```

```
        sayable.say();
    }
}
```

Reference to an Instance Method:

==========

like static methods, you can refer instance methods also. In the following example, we are describing the process of referring the instance method.

Syntax: containingObject::instanceMethodName

```
interface Sayable{
    void say();
}
public class InstanceMethodReference {
    public void saySomething(){
        System.out.println("Hello, this is non-static method.");
    }
    public static void main(String[] args) {
        InstanceMethodReference methodReference = new InstanceMethodReference(); // Creating object
        // Referring non-static method using reference
            Sayable sayable = methodReference::saySomething;
        // Calling interface method
            sayable.say();
            // Referring non-static method using anonymous object
            Sayable sayable2 = new InstanceMethodReference()::saySomething; // You can use anonymous object also
            // Calling interface method
            sayable2.say();
```

```
    }
}
```

Reference to a Constructor:

==========

You can refer a constructor by using the new keyword. Here, we are referring constructor with the help of functional interface.

Syntax: ClassName::new

```
interface Messageable{
    Message getMessage(String msg);
}
class Message{
    Message(String msg){
        System.out.print(msg);
    }
}
public class ConstructorReference {
    public static void main(String[] args) {
        Messageable hello = Message::new;
        hello.getMessage("Hello");
    }
}
```

==========

Default Methods:

==========

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

```
public interface Moveable {
   default void move(){
      System.out.println("I am moving");
   }
}
```

Moveable interface defines a method move(); and provided a default implementation as well. If any class implements this interface then it need not to implement it's own version of move() method. It can directly call instance.move();

```
public class Animal implements Moveable{
   public static void main(String[] args){
      Animal tiger = new Animal();
      tiger.move();
   }
}
```

Output: I am moving

And if class willingly wants to customize the behavior then it can provide it's own custom implementation and override the method. Now it's own custom method will be called.

```
public class Animal implements Moveable{

   public void move(){
      System.out.println("I am running");
```

```
    }

    public static void main(String[] args){
        Animal tiger = new Animal();
        tiger.move();
    }
}
```

Output: I am running

==========

Static Methods

==========

You can also define static methods inside the interface. Static methods are used to define utility methods. The following example explain, how to implement static method in interface?

```
interface Sayable{
    // default method
    default void say(){
        System.out.println("Hello, this is default method");
    }
    // Abstract method
    void sayMore(String msg);
    // static method
    static void sayLouder(String msg){
        System.out.println(msg);
    }
}
public class DefaultMethods implements Sayable{
```

```
    public void sayMore(String msg){    // implementing abstract method
        System.out.println(msg);
    }
    public static void main(String[] args) {
        DefaultMethods dm = new DefaultMethods();
        dm.say();                   // calling default method
        dm.sayMore("Work is worship");      // calling abstract method
        Sayable.sayLouder("Helloooo...");   // calling static method
    }
}
```

==========

Optional

==========

Java introduced a new class Optional in jdk8. It is a public final class and used to deal with NullPointerException in Java application. You must import java.util package to use this class.

Every Java Programmer is familiar with NullPointerException. It can crash your code. And it is very hard to avoid it without using too many null checks.

Java 8 has introduced a new class Optional in java.util package. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable because the facts which were hidden are now visible to the developer.

Optional Metods:

==========

empty(): It returns an empty Optional object. No value is present for this Optional.

of(T value) : It returns an Optional with the specified present non-null value. Otherwise it will throw the Null pointer Exception.

ofNullable(T value): It returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.

get(): If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.

isPresent(): It returns true if there is a value present, otherwise false.

orElse(T other): It returns the value if present, otherwise returns other.

orElseThrow(): It returns the contained value, if present, otherwise throw an exception to be created by the provided supplier.

public boolean equals(Object obj):

Indicates whether some other object is "equal to" this Optional or not. The other object is considered equal if:

It is also an Optional and;

Both instances have no value present or;

the present values are "equal to" each other via equals().

hashCode(): It returns the hash code value of the present value, if any, or returns 0 (zero) if no value is present.

toString(): It returns a non-empty string representation of this Optional suitable for debugging. The exact presentation format is unspecified and may vary between implementations and versions.

==========

forEach

The Java forEach() method is a utility function to iterate over a collection such as (list, set or map) and stream.

Iterating List:

```
List<String> names = Arrays.asList("Alex", "Brian", "Charles");


names.forEach(System.out::println);
```

Iterating Map:

```
Map<String, String> map = new HashMap<String, String>();


map.put("A", "Alex");
map.put("B", "Brian");
map.put("C", "Charles");


map.forEach((k, v) ->
   System.out.println("Key = " + k + ", Value = " + v));
```

Iterating Stream:

```
List<Integer> numberList = Arrays.asList(1,2,3,4,5);


Consumer<Integer> action = System.out::println;


numberList.stream()
   .filter(n -> n%2  == 0)
   .forEach( action );
```

==========

String Joiner:

==========

Java added a new final class StringJoiner in java.util package. It is used to construct a sequence of characters separated by a delimiter. Now, you can create string by passing delimiters like comma(,), hyphen(-) etc. You can also pass prefix and suffix to the char sequence.

Constructors:

StringJoiner(CharSequence delimiter): It constructs a StringJoiner with no characters in it, with no prefix or suffix, and a copy of the supplied delimiter. It throws NullPointerException if delimiter is null.

StringJoiner(CharSequence delimiter,CharSequence prefix,CharSequence suffix): It constructs a StringJoiner with no characters in it using copies of the supplied prefix, delimiter and suffix. It throws NullPointerException if prefix, delimiter, or suffix is null.

Methods:

----

add: It adds a copy of the given CharSequence value as the next element of the StringJoiner value. If newElement is null,"null" is added.

merge: It adds the contents of the given StringJoiner without prefix and suffix as the next element if it is non-empty. If the given StringJoiner is empty, the call has no effect.

length: It returns the length of the String representation of this StringJoiner.

setEmptyValue: It sets the sequence of characters to be used when determining the string representation of this StringJoiner and no elements have been added yet, that is, when it is empty.

==========
==========

Streams:

A Stream in Java 8 can be defined as a sequence of elements from a source.

Stream does not store elements. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.

The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

You can use stream to filter, collect, print, and convert from one data structure to other etc.

Java Stream vs Collection

All of us have watched online videos on Youtube. When we start watching a video, a small portion of the file is first loaded into the computer and starts playing. we don’t need to download the complete video before we start playing it. This is called streaming.

At a very high level, we can think of that small portions of the video file as a stream, and the whole video as a Collection.

At the granular level, the difference between a Collection and a Stream is to do with when the things are computed. A Collection is an in-memory data structure, which holds all the values that the data structure currently has. Every element in the Collection has to be computed before it can be added to the Collection. While a Stream is a conceptually a pipeline, in which elements are computed on demand.

Stream operations can either be executed sequentially or parallel. when performed parallelly, it is called a parallel stream.

Ways to create the Streams:

Streams can be created with the help of Stream.of(type) method.

Stream of Integer:

```
public class StreamBuilders
{
    public static void main(String[] args)
    {
        Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);
        stream.forEach(p -> System.out.println(p));
    }
}
```

Stream of Array:

```
public class StreamBuilders
{
    public static void main(String[] args)
    {
        Stream<Integer> stream = Stream.of( new Integer[]{1,2,3,4,5,6,7,8,9} );
        stream.forEach(p -> System.out.println(p));
    }
}
```

Stream Operations:

Intermediant Operations

----

map: produces a new stream after applying a function to each element of the original stream. The new stream could be of different type.

filter: The ‘filter’ method is used to eliminate elements based on a criteria.

limit: The ‘limit’ method is used to reduce the size of the stream.

sorted: The sorted() method is an intermediate operation that returns a sorted view of the stream. The elements in the stream are sorted in natural order unless we pass a custom Comparator.

flatMap: A stream can hold complex data structures like Stream<List<String>>. In cases like this, flatMap() helps us to flatten the data structure to simplify further operations

peek: peek() method returns a new Stream consisting of all the elements from the original Stream after applying a given Consumer action.Java program to use peek() API to debug the Stream operations and logging Stream elements as they are processed.

distinct: It does not take any argument and returns the distinct elements in the stream, eliminating duplicates. It uses the equals() method of the elements to decide whether two elements are equal or not

Terminal Operations:

----

forEach(): The forEach() method helps in iterating over all elements of a stream and perform some operation on each of them. The operation to be performed is passed as the lambda expression.

collect: method is used to receive elements from a steam and store them in a collection.

match(): Various matching operations can be used to check whether a given predicate matches the stream elements. All of these matching operations are terminal and return a boolean result.

count(): The count() is a terminal operation returning the number of elements in the stream as a long value.

reduce(): A reduction operation (also called as fold) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation.

toArray: We saw how we used collect() to get data out of the stream. If we need to get an array out of the stream, we can simply use toArray()

allMatch: It returns all elements of this stream which match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.

anyMatch: It returns any element of this stream that matches the provided predicate. If the stream is empty then false is returned and the predicate is not evaluated.

findFirst: findFirst() returns an Optional for the first entry in the stream; the Optional can, of course, be empty

findAny: It returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.

==========

Date And Time

==========

The new classes intended to replace Date class are LocalDate, LocalTime and LocalDateTime.

LocalDate:

****

The LocalDate class represents a date. There is no representation of a time or time-zone.

Ex:

```
LocalDate localDate = LocalDate.now();
System.out.println(localDate.toString());    //2013-05-15
```

LocalTime

****

The LocalTime class represents a time. There is no representation of a date or time-zone.

Ex:

******

```
LocalTime localTime = LocalTime.now();
System.out.println(localTime.toString()); //09:57:59.744
```

LocalDateTime

****

The LocalDateTime class represents a date-time. There is no representation of a time-zone.

Ex:

******

```
LocalDateTime localDateTime = LocalDateTime.now();
System.out.println(localDateTime.toString());      //2013-05-15T10:01:14.911
```

Instant:

****

Instant class represents an instant in time to an accuracy of nanoseconds. Operations on an Instant include comparison to another Instant and adding or subtracting a duration.

Ex:

****

```
Instant instant = Instant.now();
System.out.println(instant.toString());       //2013-05-15T05:20:08.145Z
```

Duration

****

Duration class is a whole new concept brought first time in java language. It represents the time difference between two time stamps.

Ex:

******

```
Duration duration = Duration.ofMillis(5000);
System.out.println(duration.toString());     //PT5S

duration = Duration.between(Instant.now(), Instant.now().plus(Duration.ofMinutes(10)));
System.out.println(duration.toString());  //PT10M
```

Period

****

To interact with human, you need to get bigger durations which are presented with Period class.

```
Period period = Period.ofDays(6);
System.out.println(period.toString());    //P6D
```

Date Formatting:

****

Date formatting is supported via two classes mainly i.e. DateTimeFormatterBuilder and DateTimeFormatter. DateTimeFormatterBuilder works on builder pattern to build custom patterns where as DateTimeFormatter provides necessary input in doing so.

Ex:

****

```
DateTimeFormatterBuilder formatterBuilder = new DateTimeFormatterBuilder();

formatterBuilder.append(DateTimeFormatter.ISO_LOCAL_DATE_TIME)

        .appendLiteral(&amp;quot;-&amp;quot;)

        .appendZoneOrOffsetId();


DateTimeFormatter formatter = formatterBuilder.toFormatter();

System.out.println(formatter.format(ZonedDateTime.now()));
```